# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB NO. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comment regarding this burden estimates or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

| 1. AGENCY USE ONLY *(Leave blank)* | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | 9/97 | |

4. TITLE AND SUBTITLE
Measurement and Analysis of Failures in Computer Systems

5. FUNDING NUMBERS
NASA NAG-1-613
DABT63-94-C-0045

6. AUTHOR(S)
Anshuman Thakur

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)
Coordinated Science Laboratory
University of Illinois
1308 W. Main St.
Urbana, IL 61801

8. PERFORMING ORGANIZATION REPORT NUMBER
(CRHC-97-15)
UILU-ENG-97-2222

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681

DARPA/ITO
3701 N. Fairfax Dr., Arlington, VA 22203-1714

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This thesis presents a study of software failures spanning several different releases of Tandem's NonStop-UX operating system running on Tandem Integrity S2(TMR) systems. NonStop-UX is based on UNIX System V and is fully compliant with industry standards, such as the X/Open Portability Guide, the IEEE POSIX standards, and the System V Interface Definition (SVID) extensions. In addition to providing a general UNIX interface to the hardware, the operating system has built-in recovery mechanisms and audit routines that check the consistency of the kernel data structures. The analysis is based on data on software failures and repairs collected from Tandem's product report (TPR) logs for a period exceeding three years. A TPR log is created when a customer or an internal developer observes a failure in a Tandem Integrity system. This study concentrates primarily on those TPRs that report a UNIX panic that subsequently crashes the system. Approximately 200 of the TPRs fall into this category. Approximately 50% of the failures reported are from field systems, and the rest are from the testing and development sites. It has been observed by Tandem developers that fewer cases are encountered from the field than from the test centers. Thus, the data selection mechanism has introduced a slight skew.

14. SUBJECT TERMS
measurement, fault analysis, failure analysis, fault classification, dependability

15. NUMBER IF PAGES
41

16. PRICE CODE

| 17. SECURITY CLASSIFICATION OR REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | UL |

# MEASUREMENT AND ANALYSIS OF FAILURES IN COMPUTER SYSTEMS

BY

ANSHUMAN THAKUR

B.S., University of Illinois at Chicago, 1994

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

# ACKNOWLEDGMENTS

I would like to express my gratitude to my thesis advisor, Professor Ravi K. Iyer, for his guidance and to our former group members, Dr. Inhwan Lee and Dr. Luke Young, for their contributions to this work. In addition, I would like to thank all of my friends at CRHC for answering my many questions and contributing ideas or discussion. Finally, I would like to thank my parents for their support and encouragement.

TABLE OF CONTENTS


| | Page |
|---|---|
| 1. INTRODUCTION | 1 |
| 1.1 Related Work | 3 |
| 2. THE SYSTEM ARCHITECTURE | 9 |
| 2.1 The Integrity Architecture | 9 |
| 3. DATA COLLECTION AND FORMAT | 13 |
| 4. FAULT CLASSIFICATION | 15 |
| 4.1 Broad Classification of the Failures | 15 |
| 4.1.1 Types of failure | 15 |
| 4.1.2 Final status of the failures | 16 |
| 4.1.3 Importance of the core image | 17 |
| 4.2 Detailed Fault Classification | 19 |
| 4.2.1 Panic string classification | 19 |
| 4.2.2 Detection location classification | 21 |
| 4.2.3 Fault source location classification | 22 |
| 4.2.4 Fault propagation | 24 |
| 4.2.5 Programming mistake type classification | 25 |
| 5. ANALYSIS OF TIME OF FAILURE AND REPAIRS | 28 |
| 5.1 Up-time (Age) of the System at the Time of Failure | 29 |
| 5.2 Failure Times | 30 |
| 5.3 Repair Times | 32 |
| 6. CONCLUSIONS | 35 |
| 6.1 Summary | 35 |
| 6.2 Future Work | 37 |
| REFERENCES | 38 |

# LIST OF TABLES


| Table | | Page |
|---|---|---|
| 1.1: | Measurement-based studies of computer system dependability . . . . | 3 |
| 4.1: | Classification of failures using their panic strings . . . . . | 20 |
| 4.2: | Classification of faults based on the location of the code that calls the panic routine . . . . . | 22 |
| 4.3: | Classification of failures based on faulty modules . . . . . | 23 |
| 4.4: | Classification of faults on the basis of programming mistakes . . . . . | 26 |

# LIST OF FIGURES


| Figure | | Page |
|---|---|---|
| 1.1: | Evolution of measurement-based studies of computer failure. | 4 |
| 2.1: | Tandem Integrity S2 architecture. | 10 |
| 2.2: | NonStop-UX operating system block diagram. | 11 |
| 3.1: | An example of a TPR reported against NonStop-UX. | 14 |
| 4.1: | Classification based on the nature of the failures for panics reported during the measurement period. | 16 |
| 4.2: | Status of errors reported against the OS during 1992-94, as of 1/1/95. | 17 |
| 4.3: | Importance of the core image in the repair of faults. | 18 |
| 5.1: | Software failure during measurement period starting 1992. Includes failures reported by customers only. | 29 |
| 5.2: | Age of the system at the time of crash. | 30 |
| 5.3: | Time between duplicate failures and inter-arrival times of unique faults for customer systems. | 31 |
| 5.4: | Time to repair distribution for all cases and same faults. | 33 |

# 1. INTRODUCTION

UNIX-based systems are widely used both in industry and academia. A study of failures in the UNIX operating system can enhance our understanding of software error manifestations and aid in building robust versions starting early in the software life cycle. Studies such as in [38] and [20] have investigated OS failures, but no data have been reported on failures in the UNIX operating system.

This thesis presents a study of software failures spanning several different releases of Tandem's NonStop-UX operating system running on Tandem Integrity S2(TMR) systems. NonStop-UX is based on UNIX System V and is fully compliant with industry standards, such as the X/Open Portability Guide, the IEEE POSIX standards, and the System V Interface Definition (SVID) extensions. In addition to providing a general UNIX interface to the hardware, the operating system has built-in recovery mechanisms and audit routines that check the consistency of the kernel data structures.

Our analysis is based on data on software failures and repairs collected from Tandem's product report (TPR) logs for a period exceeding three years. A TPR log is created when

a customer or an internal developer observes a failure in a Tandem Integrity system. This study concentrates primarily on those TPRs that report a UNIX panic that subsequently crashes the system. Approximately 200 of the TPRs fall into this category. Approximately 50% of the failures reported are from field systems, and the rest are from the testing and development sites. It has been observed by Tandem developers that fewer cases are encountered from the field than from the test centers. Thus, our data selection mechanism has introduced a slight skew.

In this thesis, a failure is said to occur if the system issues a panic message and crashes. A *panic* is reported by the operating system when it fails to recover from a reported error. A *duplicate failure* is defined as one failure that occurs due to the same underlying fault that was reported in an earlier failure.

Section 1.1 summarizes related research. Chapter 2 provides a description of the system architecture and the operating system. Block diagrams of both the operating system and the hardware are provided to assist the reader in associating the faults with the modules. Chapter 3 provides details about the TPRs. An example of a TPR has also been provided. Chapter 4 concentrates on classification of failures based on the panic strings, the detection location of the error, the source location of the fault, and the type of programming mistakes committed during development. Chapter 5 brings into perspective the fault behavior of the software for the period extending from 1992 through 1994 and provides failure timing distributions, which include the distribution of time between failures and the distribution of time taken to repair the fault. Chapter 6

Table 1.1: Measurement-based studies of computer system dependability

| Category | Issues | Studies |
|---|---|---|
| Data Coalescing | Analysis of time-based tuples | [44], [8] |
| | Clustering based on type and time | [15], [19] [43] |
| Basic Error Characteristics | Transient faults/errors | [36], [29], [15] |
| | Error/failure bursts | [15], [9], [43] |
| | TTE/TTF distributions | [29], [14], [21] |
| | Fault window and impact of failures | [5] |
| | Fault latency | [3], [49], [30] |
| Dependency Analysis | Hardware failure/workload dependency | [1], [2], [12] |
| | Software failure/workload dependency | [2], [14], [34] |
| | Correlated failures and impact | [39], [48], [42] |
| | Two-way and multiway failure dependency | [6], [19], [40] |
| Modeling and Evaluation | Performability model for single machine | [10] |
| | Model for distributed systems | [43], [11] |
| | Two-level models for operating systems | [21] |
| Software Dependability | Error recovery | [47], [9] |
| | Hardware-related and correlated software errors | [13], [42], [21] |
| | Software fault tolerance | [7], [20], [22] |
| | Software fault tolerance using software diversity | [28] |
| | Software defect classification | [37], [38], [4] |
| Operational Security | Experimental data from controlled experiment and definition of security measures | [26] |
| Fault Diagnosis | Heuristic trend analysis | [44], [25] |
| | Statistical analysis of symptoms | [16] |
| | Network fault signature | [31], [32], [33] |

concludes the thesis by summarizing the important results of this research and laying out the plans for future work.

## 1.1 Related Work

Measurement and analysis of computer systems started in the late 1970s. Figure 1.1 traces the evolution of measurement-based studies. It associates measurement-based research studies with five distinct time periods. Each period consists of five years. We

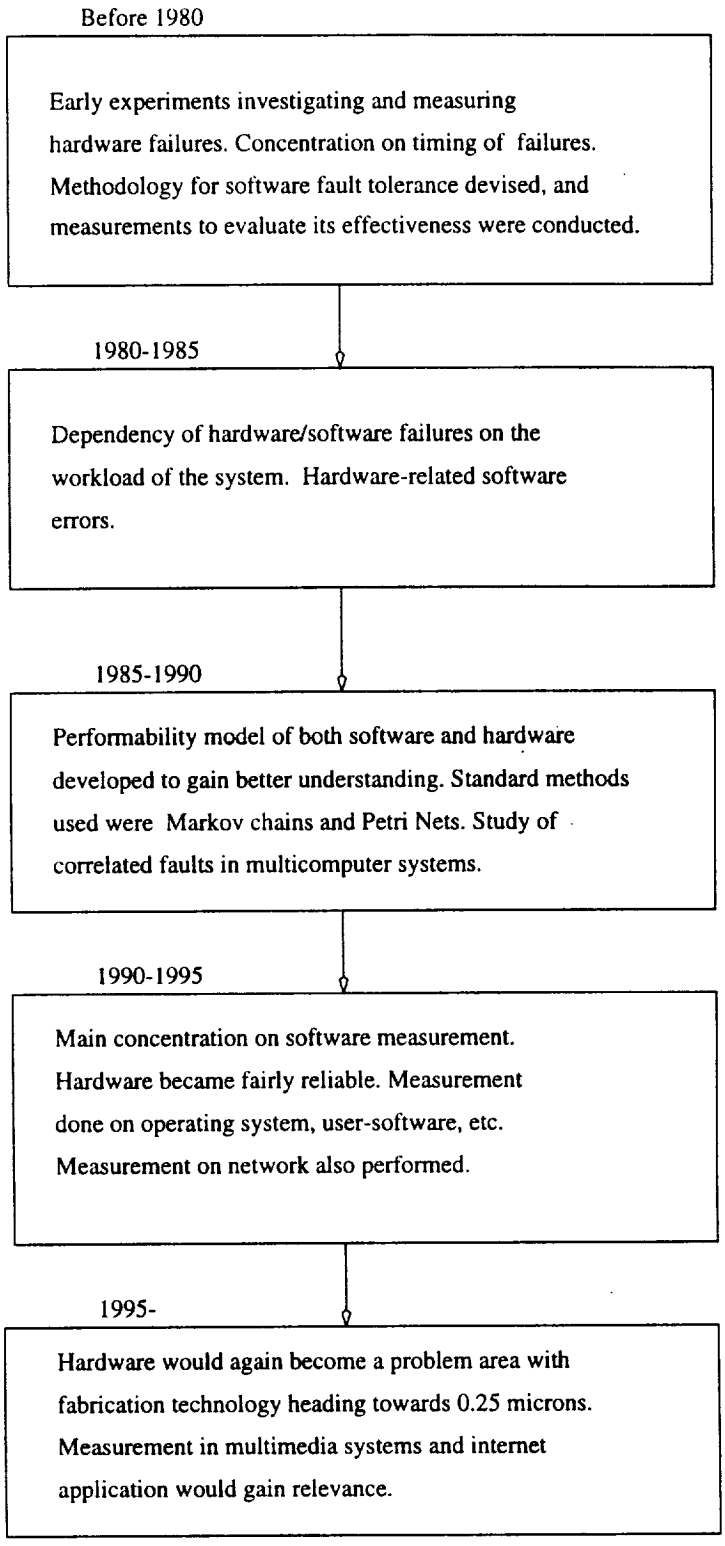

Before 1980
Early experiments investigating and measuring hardware failures. Concentration on timing of failures. Methodology for software fault tolerance devised, and measurements to evaluate its effectiveness were conducted.
1980-1985
Dependency of hardware/software failures on the workload of the system. Hardware-related software errors.
1985-1990
Performability model of both software and hardware developed to gain better understanding. Standard methods used were Markov chains and Petri Nets. Study of correlated faults in multicomputer systems.
1990-1995
Main concentration on software measurement. Hardware became fairly reliable. Measurement done on operating system, user-software, etc. Measurement on network also performed.
1995-
Hardware would again become a problem area with fabrication technology heading towards 0.25 microns. Measurement in multimedia systems and internet application would gain relevance.

Figure 1.1: Evolution of measurement-based studies of computer failure.

can see from the figure that the nature of research work is migrating from hardware to software. Researchers have put more stress on software failures in the recent years because it has been found that software errors are responsible for 60-70% of system failures. Table 1.1, adapted from [17], has been updated and provides classification of major research work in the measurement area. Next, we will in detail discuss some of the closely related work.

The issue of software reliability has been studied extensively, and a large number of models have been proposed [27], [35]. For the most part, these models attempt to describe the reliability growth of the software during the development phase.

Early measurement-based studies of operational software running on the IBM 3081 and DEC systems in [14] and [2] showed the strong correlation between the workload of the system and the failures. The research in [2] introduced a workload-dependent probabilistic model to predict the differences in manifestations of hardware transient and software errors as a function of system workload. The model was applied to a modified version of the standard TOPS-10 operating system running on a PDP-10 machine.

Analysis of failures of the VM/CMS and the MVS/SP operating system running on the IBM 3081 machine [13], [14] found that 25% to 35% of all software failures were hardware-related. Storage management and exceptions were identified to be the prime cause of software errors.

A Markov model that describes software errors and the recovery process in a production environment using error logs from the MVS operating system was discussed in [9].

The model developed using low-level error data provides a quantification of the system error characteristics and the interaction between different types of errors. It was found that multiple errors constituted 17% of all software errors and involved high-recovery overhead.

Software errors on the VAX/VMS operating system in VAX cluster multicomputer environments were analyzed using error logs in [41]. The correlations between software and hardware errors and among software errors on different machines were investigated. The results showed that most software problems were from program flow control and I/O management. It was concluded that network-related software was the reliability bottleneck. The distribution of time between errors was modeled by a two-phase hyper-exponential random variable.

In [7], the results from a census of Tandem systems were presented. It was found that the number of hardware failures has reduced considerably with time, and the software errors account for 62% of the outages.

Software defects in the MVS operating system have been studied using the software error reports in [38]. Three parameters, the error type, the defect type, and the error trigger, have been used to classify errors reported from the DB2 and the IMS product. It has been shown that the error characteristics of different products were different. A detailed discussion of the undefined state of software (when it fails) has also been provided.

The use of the observed software defects to provide feedback to the development process is discussed in [4]. Software defects are mapped back to the different phases of software development. A deviation from the standard defect pattern would indicate an imperfect software development procedure. It has been shown that functional errors are predominant in the early developmental phase, whereas timing/synchronization errors are more common during the test phase.

In [5], a study of the impact of failures on customers and the fault lifetimes was presented using failure data from commercial software. Two metrics, fault weight(which is based on the severity of the fault) and fault window (which is the time frame in which the fault was reported), were introduced.

In [20] and [24], a study of the Tandem GUARDIAN operating system was presented. Data collected by a system monitoring facility and the TPRs reported against GUARDIAN were used to conduct the analysis. The failure times and the classification of errors were presented. The error propagation study conducted revealed that 47% of the errors had a short error latency, whereas 41% showed significant error latency. Most of the errors were detected quickly without corrupting other segments of the code.

A wide-ranging analysis using error classification of software error data collected during the development phase was reported in [46]. This study has important results. First, it provides comprehensive data on UNIX failures (although the UNIX system considered is significantly more robust/dependable than a typical UNIX system). Second, it relates the symptoms of the failures to where the error was detected (the panic invocation point),

the source of the error (the module to be fixed), and the panic string that was generated when the error was detected. It also presents the most prevalent kinds of programming mistakes in the software. Third, it provides a comprehensive study of failure timing distributions. Finally, by combining systems from the internal high-stress environment and systems from the field sites, it provides a wide-ranging view of the system.

# 2. THE SYSTEM ARCHITECTURE

## 2.1 The Integrity Architecture

The NonStop-UX operating system runs on the Tandem Integrity S2 architecture. Figure 2.1 provides a block diagram of the architecture. The system consists of a triplicated processor local memory system contained on three Central Processing Units (CPUs). Duplexed Triple Modular Redundant Controllers (TMRCs) provide a large secondary main memory and serve as the nexus for the I/O operations of the machine. This is a major center for error detection, as all data transfers are checked. The TMRC also does the voting in the architecture. The CPU/local memory connects to the TMRCs over the Reliable System Bus (RSB). Duplexed Input/Output Packetizers (IOPs) provide the interface between a TMRC on one side and a superset of an industry-standard I/O bus (VME) on the other. The interconnection between the IOPs and the TMRCs is called the Reliable I/O Bus (RIOB). The IOPs are the conduit through which all I/O in the machine flows. Each IOP controls a bus called the NonStop-V+ bus, which is the high

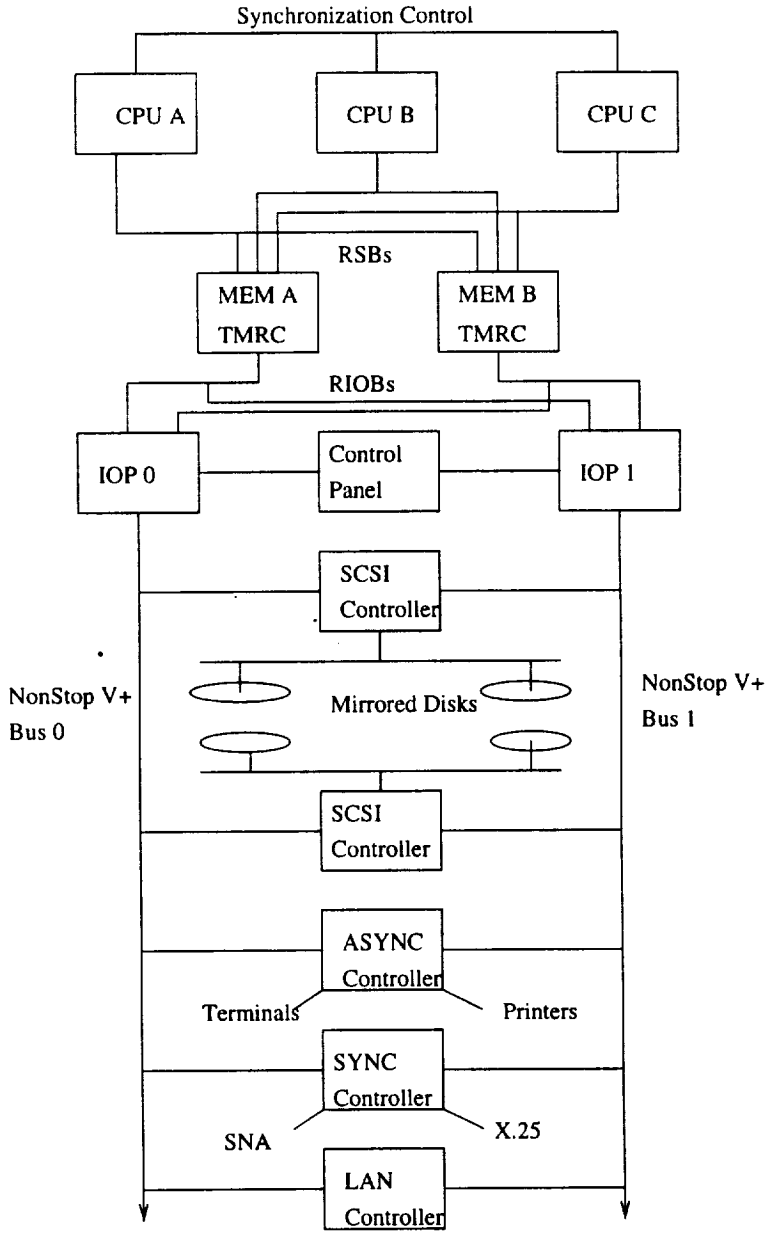



Figure 2.1: Tandem Integrity S2 architecture.

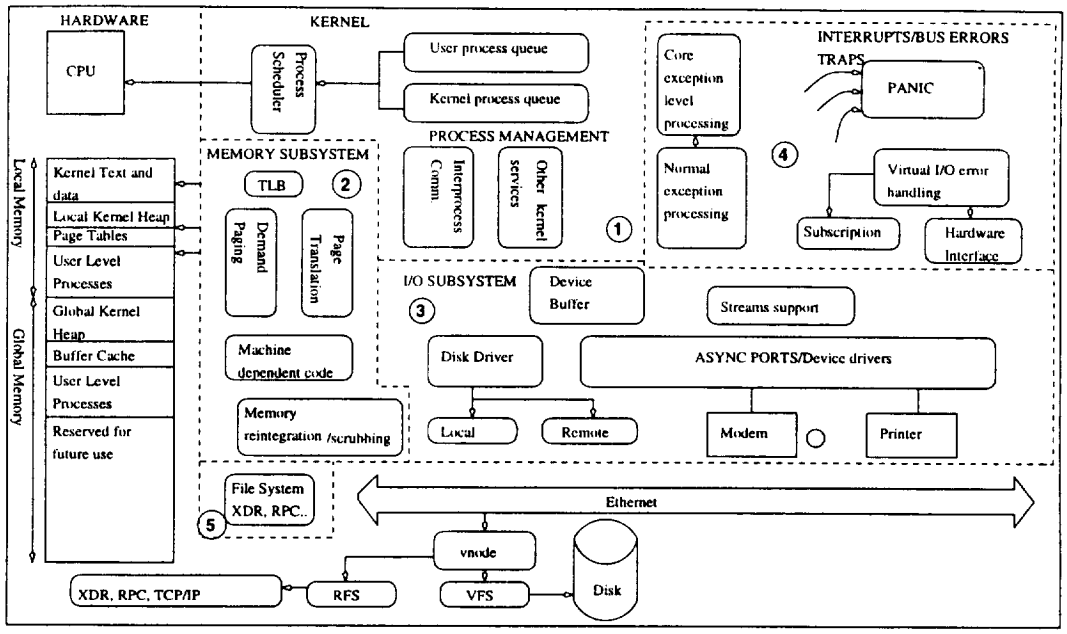



Figure 2.2: NonStop-UX operating system block diagram.

data integrity version of the VME bus. To connect an ordinary VME controller to the system, a Bus Interface Module (BIM) is connected to the VME controller, and the BIM connects to the NonStop-V+. The BIM provides a dual-ported path from the peripheral controller to each IOP. Peripherals are connected to the VME controllers via the Patch Panel.

The reader is assumed to be familiar with UNIX System V; thus, we will only mention some of NonStop-UX's unique features. Figure 2.2 provides a block diagram of the OS. The local memory (present on the CPU board) contains the kernel text and data, a local kernel heap, page tables, and user processes. A similar layout exists for the global memory (physically present on the TMRCs). If data are not found in the local memory, they are fetched from the global memory.

In addition to standard error-handling mechanisms such as signals, techniques are used to enhance the robustness of the OS. These techniques include a highly reliable enhanced kernel that corrects High Data-Integrity Panics (HDIPs) and uses audit routines for checking the consistency of kernel data structures. Thousands of assertions that check for a viable condition, such as correct pointer values, have been inserted throughout the OS code. Recovery routines provide mechanisms for problem isolation and for preventing system corruption by killing problematic processes. Facilities, such as an indestructible initialization (init) process and a keepalive daemon, allow the system to survive termination of vital operation processes. If a crucial process is terminated, these facilities restart it. The data are protected from corruption by using disk checksums, providing strong tolerance to power loss, and duplicating data across multiple disks. There are many other hardware robustness mechanisms that support OS recovery. Most of these mechanisms are covered in [18].

# 3. DATA COLLECTION AND FORMAT

A Tandem Product Report (TPR) logs the occurrence of a failure and tracks in detail the problem diagnosis and repair. TPRs form the basis of our study. Typically, when a failure occurs at a user's site or through extensive testing during the development of a release, it is subsequently logged, analyzed, and corrected. In some cases, the symptoms provide sufficient information to determine the cause of the failure (generally in cases of duplicates), in which case the data are not entered in the TPR database. If, however, the developer concludes that further investigation is needed before a conclusion can be reached, then the failure is entered into the database.

Figure 3.1 provides an example of a TPR reported for a failure in a NonStop-UX system. The header section includes the date and time of the TPR creation, severity/priority of the reported error, and the current status of the error. The second section provides textual descriptions of the problem, the analysis procedures undertaken to correct the fault, and the fix information. Although not shown in the example, some TPRs also contain output from runs of *crash* on the memory core.

PRODUCT REPORT

Date Reported: xx-xx-xx Time of report: xx:xx Type of Problem: HW/SW/Unknown

Severity level: A-C Customer information: Substatus of error: xx

Description of problem: Machine panic'ed with an assertion failure. (Assertion XYZ checking pointer value P failed.) Before the machine panic'ed, the UNIX command "ping" stopped working, all telnet sessions were hung, and the machine complained that the ethernet driver buffer was full. Both the console messages and the core are available.

Analysis procedures/results: The error was located in the ethernet driver, and the driver was modified in the following way. Open requests are allowed to the ethernet driver even if the controller is offline. This allows the streams pointer to always be in a valid state. Earlier, if the device was offline, the open did not succeed; however, the streams pointer was used subsequently.

-Analyst xxx (Dated: xx-xx-xx)

Fix incorporated in the affected module on xx-xx-xx by xxx.

Figure 3.1: An example of a TPR reported against NonStop-UX.

Two kinds of data were extracted from the TPRs for our study. The first relates to the failure detection and repair process. These data include detailed information about the analysis procedures undertaken at Tandem and the fix information, if the problem was fixed. Analysis procedures illustrate the use of core image and console messages and provide chronological listings of the procedures carried out to locate and fix the error.

The second kind of data consists of the times at which problems were reported and at which they were fixed. In the case of duplicates, the fix time is the time at which it was ascertained that the problem was a duplicate.

# 4. FAULT CLASSIFICATION

## 4.1 Broad Classification of the Failures

### 4.1.1 Types of failure

Figure 4.1 provides a high-level summary of the types of failures reported during the three-year period. These failures fall into three categories. Sixty-three percent of the failures were confirmed to be software related, and 4% of the failures were hardware related. In 33% of the cases, the failure could not be classified; failures in this category were generally problems that could not be fixed or problems in which changing both hardware and software components corrected the problem. In the latter case, the cause of the failure remained unknown. In [24], of all the failures reported in the GUARDIAN operating system, 89% were software related, and the rest were due to hardware faults.

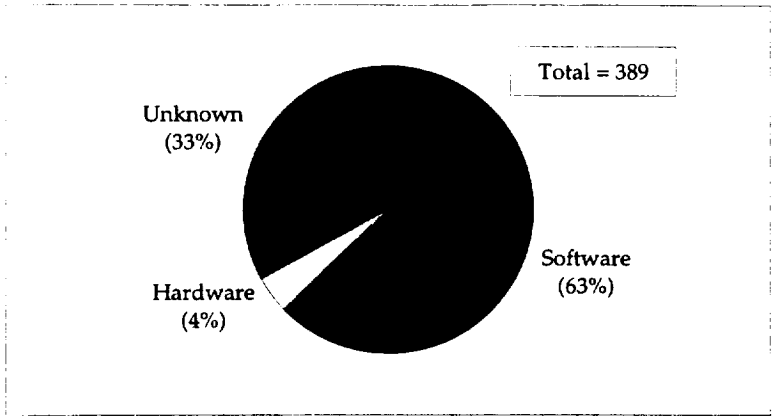



Figure 4.1: Classification based on the nature of the failures for panics reported during the measurement period.

### 4.1.2 Final status of the failures

Many of the failures reported were caused by the same fault as a failure reported earlier. Figure 4.2 shows the breakdown of the final status of all of the failures reported during the three-year period against various releases of the NonStop-UX operating system. Forty-three percent of failures during this period were duplicates. This can be compared with the 76% and 47% re-occurrence proportion reported in release 1 and release 2 of the software in [5], and 72% in [24]. Twenty-eight percent (in the category *Tested*) were unique problems, which were fixed and passed through quality assurance. In the absence of a core image and console messages, an effort was sometimes made to recreate the problem. The core image and console messages were recorded if the problem could be recreated. Eighteen percent of the failures, however, could not be recreated on Tandem test machines. There are several explanations for this. 1) The system used for testing may not have been the same as the system that failed (although all efforts

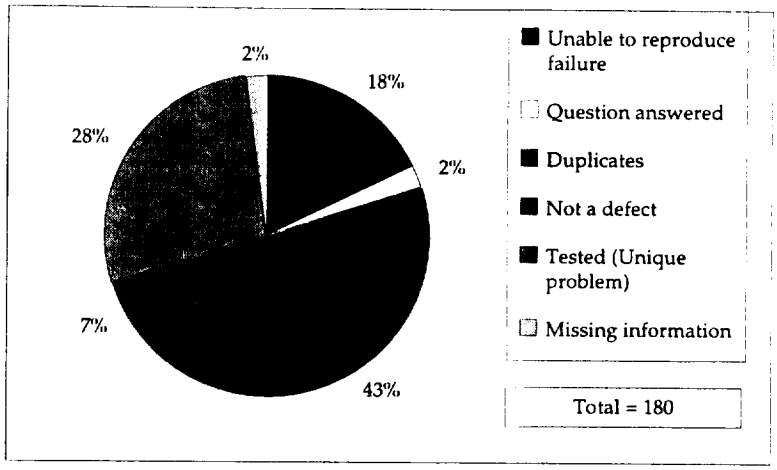



Figure 4.2: Status of errors reported against the OS during 1992-94, as of 1/1/95.

are made to recreate the failure environment). 2) Stress conditions may not have been the same. 3) Other software interactions with the OS at the time of failure may not have been present. A reported failure is classified as "not a defect" when the fault is due to user misunderstanding (e.g., when the user fails to follow the manual and installs incompatible OS modules).

### 4.1.3 Importance of the core image

Of all of the failures, only 79% supplied the memory core image. In the other 21%, a core image either was not created by the system when it failed, was lost during transportation, or was rendered unreadable when it arrived at Tandem. Systems failed to save a core image when there was insufficient disk space to store the core image or in a few cases when double panics occurred. A *double panic* may occur, for example, when a system encounters an inconsistency while trying to *sync* the filesystem inside the panic routine. Nine percent of all failures reported in the OS were double panics.

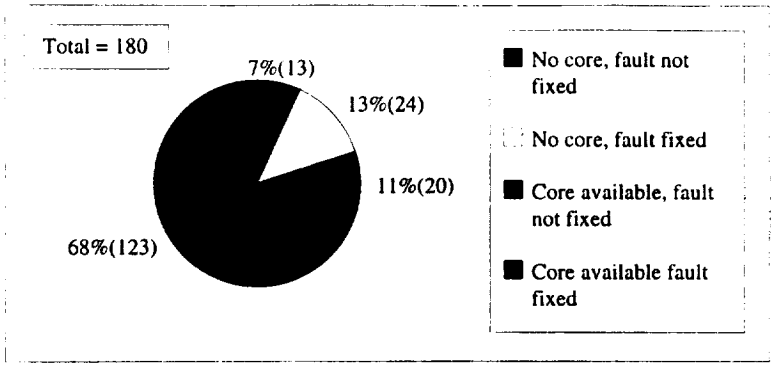



Figure 4.3: Importance of the core image in the repair of faults.

Availability of a core image is very important to the fault repair process. Using the core image, the analyst can see the various operating system internal tables and the memory structures. Generally, the procedure stack of the processes that were running at the time of failure provides some clue about the location of the fault. Lee, Iyer, and Mehta [23] discuss the use of a procedure stack in the identification and correction of software failures. The file table, process table, and callout table also aid in fault identification and location. Figure 4.3 shows the importance of the core image in the location and correction of faults. We see that of the 143 failures containing core images, only 14% (20 out of 143) could not be fixed, whereas of the other 37, for which no core image was available, 35% (13 out of 37) of the errors could not be corrected. The critical importance of the core image in the repair of a fault makes it clear that more effort must be made to provide a better core-saving mechanism. Static allocation of disk space and regular removal of old core images from that partition by the administrator can alleviate the problem of failure to save the core image due to insufficient disk space. In the event of a disk space shortage in a small system in which a large reserved partition on a disk would

not be feasible, a partial core consisting of the major internal tables and the process stack could be saved.

## 4.2 Detailed Fault Classification

In this section, we describe some fault classifications to provide more information relating fault source location to the detection location. The classifications presented also shed light on programming mistakes in the OS. In deciding on classes, our aim was to abstract at a level that would facilitate the tabulation of failures, while preserving relevant details.

The general system behavior during a failure can be characterized as the following sequence of events. A system encounters an error situation, such as the failure of an assertion, and then passes control to an error-handling routine if that assertion has an associated error handler; otherwise, it panics. The error-handling routine determines whether the reported error is recoverable. If the system cannot recover from the error, it then calls the panic routine, which starts the panic process. The panic process attempts to save the core image, displays a panic string on the console, and initiates an automatic reboot.

### 4.2.1 Panic string classification

The *panic string* displayed on the console provides an error message describing the last action that crashed the system and initiated the panic. Table 4.1 displays the failures

Table 4.1: Classification of failures using their panic strings

| Panic String Displayed | Number of Failures Total 86 | Percent of total (rounded off) |
|---|---|---|
| Assertion failed | 32 | 37.2 |
| Translation Lookaside Buffer miss | 12 | 14 |
| Bus error | 4 | 4.7 |
| Recovery not allowed | 11 | 12.8 |
| Recovery threshold exceeded | 3 | 3.5 |
| Streams recovery handler timed out | 2 | 2.3 |
| Kernel trap | 2 | 2.3 |
| Kernel memory freed | 2 | 2.3 |
| Kernel memory coalesced | 1 | 1.2 |
| Kernel stack overflow | 3 | 3.5 |
| Interrupt stack overflow | 3 | 3.5 |
| TMRC data transfer | 3 | 3.5 |
| Fatal TMRC error | 1 | 1.2 |
| Swap failed in user area | 2 | 2.3 |
| Page substitution | 1 | 1.2 |
| Hardware specific memory management problem | 1 | 1.2 |
| Unexpected write I/O failure | 2 | 2.3 |
| Byzantine data in writeprotect RAM | 1 | 1.2 |

classified by the panic string. It should be noted that a panic string was not available for all of the failures, and, thus, only a subset of the TPR data was used for the classification.

We see that about 40% of the panic strings displayed the message *assertion failed.* This directly suggests that assertions provide good error detection. Assertions test a condition, such as the value of a pointer. If the value does not match the expected value, the assertion transfers control to the recovery routine or calls a panic.

The next major contributor was the error message *TLB miss in kernel space.* This occurs when the kernel references an address pointer that has no mapping in the TLB. A wrong address for the kernel page is usually generated when a pointer data structure in the kernel is corrupted.

The fourth row of Table 4.1 shows that in 15 cases the OS entered the recovery routines, and then, after several trials, the recovery routine declared the error unrecoverable and invoked a panic. Other failures reported include kernel memory errors, errors during transfer of data from the TMRC (global memory), some I/O errors, and one Byzantine failure.

### 4.2.2 Detection location classification

Although the discussion so far suggests that many errors were detected by the assertion tests, it does not reveal much about the propagation already incurred. Table 4.2 provides a classification based on the location of the code that called the panic routine. Most of the errors were detected in the machine-dependent virtual memory module, in which an assertion failed and the panic routine was called. A probable implication of this is that these errors have incurred some propagation because the machine-dependent code for virtual memory resides at a very low level. Many errors were also detected in the streams module. These errors were due to software defects in the streams module itself and to the propagation of device driver faults into the streams module when the streams mechanism was used by device drivers for communication. The third row in

Table 4.2: Classification of faults based on the location of the code that calls the panic routine

| Code section from which invokes panic | # failures Total 92 | Fig.2 map |
|---|---|---|
| Virtual memory management | 8 | 2 |
| Machine dependent virtual memory and process management | 23 | 2 |
| Streams subsystem | 14 | 2 |
| File system operations | 7 | 5 |
| Asynchronous device driver | 10 | 3 |
| Related to dispatch queue.(process scheduling, process preemption, context/process switching) | 3 | 1 |
| Sleep-wakeup hashing | 2 | 1 |
| V node operations (read/write/release/open/create) | 4 | 5 |
| Transport Interface Library R/W Module | 1 | 3 |
| Routines for moving data around (used by device drivers) | 1 | 3 |
| Remote procedure call section | 1 | 5 |
| Diagnostic/Integration section (memory scrubbing/re-integration) | 4 | 4 |
| Error message logger | 14 | 4 |

Table 4.3 gives the number of defects encountered in the streams subsystem. Similar reasons explain the phenomenon observed in the asynchronous device drivers, which have a significantly high incidence of software defects. A number of *panic* calls were made from the error message logger, which is responsible for passing messages to *syslogd*. In all cases, it encountered a TLB miss and then panic'ed.

### 4.2.3 Fault source location classification

Table 4.3 shows the number of software defects identified in each module. We separate the number of unique faults from the total number of failures to show the visibility of

Table 4.3: Classification of failures based on faulty modules

| Module affected | No. of unique failures | Total | Fig.2 map |
|---|---|---|---|
| Device drivers (async, ethernet, etc.) | 15 | 31 | 3 |
| Memory subsystem | 8 | 16 | 2 |
| Streams mechanism | 5 | 12 | 3 |
| Process management | 3 | 6 | 1 |
| Machine dependent VM code | 4 | 8 | 2 |
| Shutdown/Bootup process | 4 | 8 | not shown |
| Filesystem | 6 | 10 | 5 |
| I/O subsystem related to data movement | 3 | 3 | 3 |
| Mirror driver | 1 | 1 | 3 |
| Interrupt handling | 1 | 1 | 4 |
| Diagnostic/Integration | 2 | 3 | 4 |
| MIDAS(monitoring facility) | 1 | 1 | not shown |
| Total | 53 | 100 | |

each group of faults. If the proportion of duplicates $((total - unique)/total)$ is high, as in the case of the device drivers and the memory subsystem, there is a direct implication that faults in this area will affect many users and should be repaired immediately.

Table 4.3 also shows that the device drivers (which include disk, other async, and ethernet drivers) contribute to the greatest number of faults. This result is mainly because the system interacts with the outside world through these device drivers, and many exceptions occur at this level. Many of the faults located in this region of the code were missing checks for some exception. Some of the faults were also due to request conflicts or to exceeding the limit of the number of *opens* in the device drivers.

Faults in the streams section of the code involved freeing a memory location or making a pointer NULL and then trying to use it. In some cases, exception situations were not checked.

Filesystem faults usually occur when the system is working under a high load or in unrecommended situations, such as booting up the system with some disk removed. Although the booting process is generally robust, unanticipated exceptions were encountered. Ten filesystem failures were reported during the period, of which four were duplicates.

Faults in the page-swapping mechanism and faulty allocation and deallocation of memory when forking a process are typical examples of memory subsystem faults. Eight unique faults were reported in the memory subsystem.

The information in Table 4.3 can be used to direct major code correction efforts in modules, such as the device drivers and in the memory subsystem, because not only are these modules more likely to have faults, but also the faults in these regions have high visibility (manifest into errors easily).

### 4.2.4 Fault propagation

Relating the classifications in Tables 4.2 and 4.3, we derive the propagation incurred by the faults before they were detected. Although the number of TPRs providing information for both the tables has been reduced slightly, we find that 62% of the errors were

detected in close proximity to the location where they were generated. By "close proximity" we mean that the errors were detected in the code that performed similar actions and that these faults were present in the same subsystem. The other 38% propagated to unrelated code sections. Many of the errors caused by faults from the streams subsystem were detected in the machine-dependent section when an assertion failed. These problems generally occurred when a wrong address in a pointer subsequently interacted with the memory. With 38% of the faults incurring some propagation before detection, there is room for improved error containment. Building in fault tolerance with object-oriented languages might reduce propagation. In addition to doing a check when a value is used, a consistency check when a value is written into a structure would result in instant detection. However, there is certainly an issue of the overhead involved.

### 4.2.5 Programming mistake type classification

Lastly, the type of programming mistakes committed during the development of the OS was made the basis of classification. Table 4.4 provides the details of the classification. In this table, we have left out the duplicates, because those do not provide any additional information.

The first category relates to mistakes made in pointer data structure management. "Missing check for an exception" means that an *if* statement to check an obvious check condition was missing. The "incorrect logic" part groups all failures due to an incorrect algorithm in the code or to wrong code placement. This class does not include cases in

Table 4.4: Classification of faults on the basis of programming mistakes

| Cause | Number of Cases |
|---|---|
| Pointer made NULL and later used | 9 |
| Pointer assigned to wrong location | 5 |
| Stale pointer left from before | 1 |
| Missing check for an exception | 14 |
| Incorrect algorithm or code placement (includes major algorithm mistakes) | 14 |
| Uninitialized data structures | 2 |
| Memory allocation/deallocation | 6 |
| Unnecessary code left in the OS | 2 |
| Total | 53 |

which trivial statements, such as exception checks or pointer tests, were overlooked. A mistake in which a wrong assumption about the timing was made and about where the code is placed at a wrong location would fall into this category. The "uninitialized data structures" category is used to group failures in which a data structure is used before its initialization. "Memory allocation/deallocation errors" constitutes the next group and includes errors such as memory leaks or freeing memory at the wrong time. "Unnecessary code" is the category that is not seen very often. This class means that some code (e.g., debug code) that was not required for proper functioning was left in the OS. A similar classification has been used in [4]. Many of the errors involved pointers that either became NULL before their use was over or were pointing to the wrong location. As is common to all software, variables were sometimes uninitialized. At places, the semantics of the code did not represent the intended algorithm.

Because many mistakes were due to mismanagement of pointer data structures, a more robust methodology for program implementation should be followed. Research in [45] suggests that using data structures, such as linked lists with backward pointers that have inherent error detection and correction capabilities, might improve overall software robustness. A feasibility study of robust data structures and their effect on software performance would provide suggestions on their usability.

# 5. ANALYSIS OF TIME OF FAILURE AND REPAIRS

Time of failure and repair also portray failure characteristics. Figure 5.1 gives the distribution of failures encountered on customer systems during the measurement period. The x-axis starts at the beginning of the year 1992. Although the first version of the operating system was released in 1990, not much data are available for the period 1990-91. This is partially because there were not many customers, and the data collection system was unorganized. During the first six months there was not much activity. After 200 days, the failures reported increased as a major customer started testing the system and contributed to the failure reporting. Discussions with developers at Tandem helped us to identify the various factors that affect the failure reporting rate. Those factors are: 1) an increase in the number of customers, 2) a diversification of the customer base resulting in usage of the OS in many different ways, and 3) an increase in the number of supported features (Tandem adds certain features to the OS for major customers). Despite the involvement of the above factors, we see that the failure reporting frequency

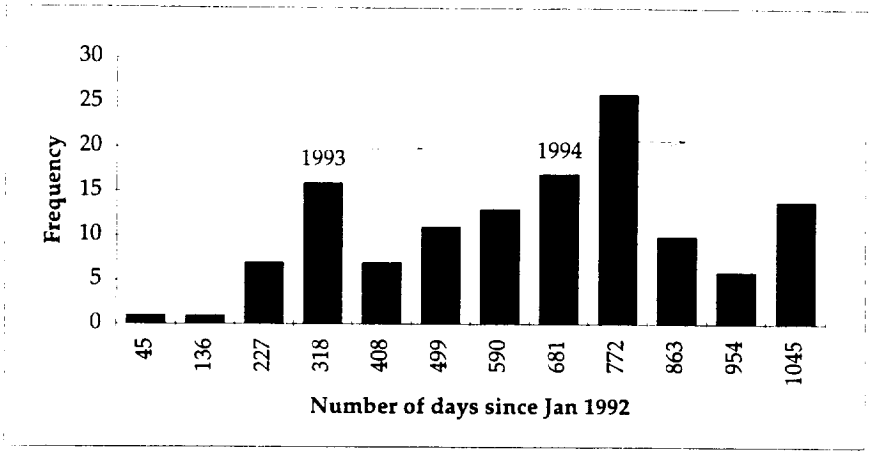



Figure 5.1: Software failure during measurement period starting 1992. Includes failures reported by customers only.

has decreased, suggesting that the OS has matured considerably. During this period, Tandem released more than 50 major and sub-releases of the operating system.

## 5.1 Up-time (Age) of the System at the Time of Failure

*Up-time* of a system is defined as the time elapsed since the last bootup of the system. Sometimes this is also called the *age* of the system. Figure 5.2 shows the age distribution of the customer systems at the time of failure. In many cases, when a new device is installed, the kernel has to be recompiled with new parameters. Generally, the failures reported with a small system age occur after this installation. The installation of new devices, for example, uncovers faults in sections of code that were never executed before. Incorrect setup of the newly installed devices can crash the system. In some cases, the same fault crashed the system multiple times before a remedy could be supplied by Tandem. The crash usually occurred because the user wanted to perform a specific task

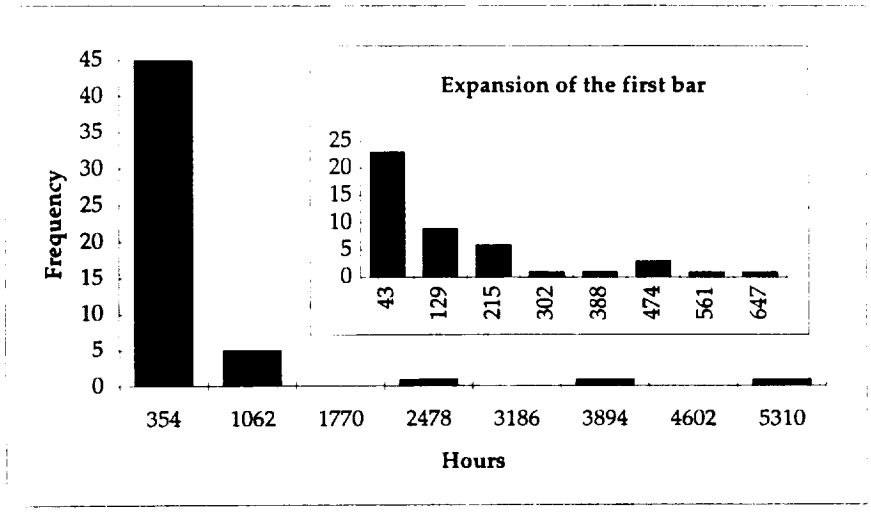
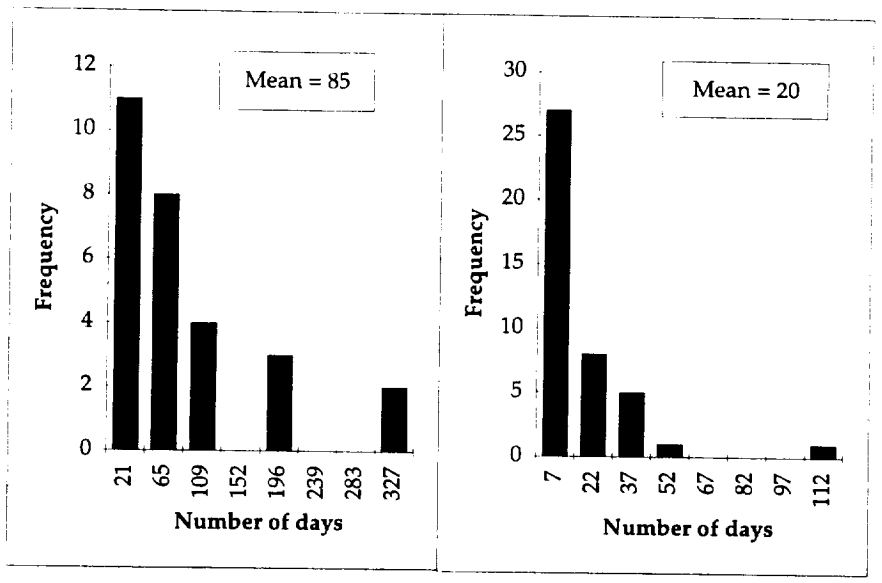



Figure 5.2: Age of the system at the time of crash.

repeatedly and encountered the same errors in quick succession. This result corroborates the findings in [2] and [41]; however, in our case, the up-time period was longer because we have only looked at major failures (panics).

## 5.2 Failure Times

In this section, we provide the distribution of the time between unique failures and the duplicate failures for customer systems. The TPR creation time has been used to approximate the failure time. The usage of TPR time as an approximation of failure time introduces some uncertainty about the actual numbers obtained from the distribution. It was observed that there was always a delay between the failure time and the failure report time because of the delays at the customer's end and the standard delay in the TPR entry procedure. For some cases in which information about the actual failure time




(a) TBF of same faults (b) Inter-arrival of unique faults

Figure 5.3: Time between duplicate failures and inter-arrival times of unique faults for customer systems.

was available, we compared the actual failure time to the TPR time, and an average delay of 14 days was found. The variance was comparatively small.

Time between failures for all systems, although a useful parameter in cases in which all systems are uniform, does not provide useful information in our case because systems under consideration can be using different versions of the OS. It was observed that many customers did not upgrade the OS for a long time, primarily because of the fear that installing something new might have detrimental effects on their system's operation.

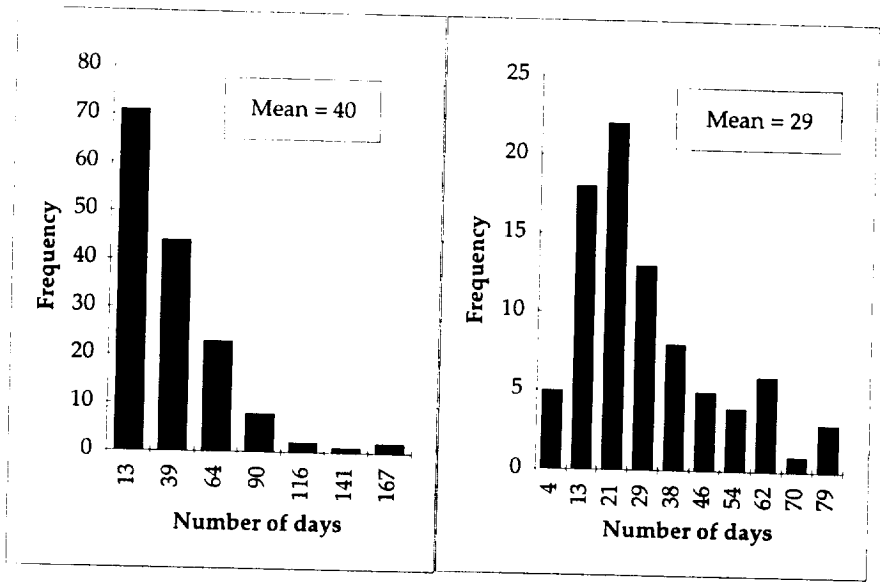
However, we provide two failure-time distributions that are useful in understanding the behavior of the failures. Figure 5.3(a) provides the distribution of time between failures due to the same fault. It was obtained by subtracting the failure time of each of the duplicate faults from the failure time of its first reporting. Similar timing distribution

has been reported in [5]. Many faults having the same cause were reported in quick succession. This may have been due to either of the two following reasons 1) a new version of OS was supplied by Tandem simultaneously and, thus, failures are reported around the same time, or 2) the same customer reported the same error more than once before that problem was corrected. Duplicates have been reported sometimes after a very long period (300+ days). Many customers waited a long time before up-grading to new versions of the OS, and the rediscovery of an old fault was reported after a long time. The mean time recorded between duplicate failures was 85 days.

Figure 5.3(b) gives the distribution of the inter-arrival times of the unique failures. A failure that is not a duplicate of a fault reported earlier is called *unique*. The mean time was 20 days.

## 5.3 Repair Times

When a system crashes, it generally comes up after a reboot if the crash was not due to a software defect present in the bootup module. We do not define repair time as the time to bring the system back into a running state, but as the difference between the time of failure and the time at which the problem was identified and corrected. The problem could either be a new problem or just a recurrence of a previously reported fault. In the case of a fault that is a recurrence of a previously diagnosed fault, the repair time is the time taken by the Tandem developers to ascertain that the fault was indeed a duplicate.




(a) All faults (b) Duplicates

Figure 5.4: Time to repair distribution for all cases and same faults.

The calculation of the repair time was done by subtracting the time at which the fault was entered from the time at which a final conclusion was drawn. We then add the mean of the difference between the time of failure and the TPR time (which was 14 days) to extend the period to the time of the crash. Figure 5.4(a) gives the distribution of the time to repair all (testing/customer) failures. The mean time to repair was 40 days. This time includes the delay in reporting the failures and the time taken by the customer to send the details of the failure, such as console messages and memory core image. The first bar in Figure 5.4(b) (with repair time of four days) consists of those cases in which a problem was resolved even before the TPR was assigned to a developer. The longer times resulted when there was a delay in sending the information. It should be noted, however, that in many cases a workaround or a temporary fix (not fully tested) is provided to the customer very quickly (long before the case is closed). There were many cases in which

the core image sent by the user was unreadable, or it reflected a different machine state (wrong core file). Repair time also went up when the problem could not be reproduced in the test laboratory. In some cases, a fault thought to have been repaired was closed, and later, it was discovered that the suggested repair did not eliminate the problem.

Usually the duplicate errors were repaired in less time because both the detection and the correction of the problem were easier. First, the duplicate failures generally show similar symptoms to those in the previously reported cases [23]. Second, because the fix is already available, the implementation time is eliminated. Most of the faults were repaired within a month. Figure 5.4(b) gives the distribution of the time to repair the duplicates with a mean of 29 days. The duplicates that can be easily identified by the first-or second-line support people are not documented as the TPRs. In these cases, the repair time is fairly short. Figure 5.4(b) does not show such duplicates.

# 6. CONCLUSIONS

## 6.1 Summary

With the widespread use of UNIX systems today, there is an urgent need to form a clear understanding of its dependability. In this thesis, we have presented an analysis of failures in the various releases of the NonStop-UX operating system, which is based on UNIX System V. Initial classifications were based on the nature (hardware/software) and the current status of the failures. An in-depth fault classification was provided. We classified faults based on their source location, their detection location, the panic string displayed on the console, and the kind of mistakes that were made by the developers of the OS. We also provide distributions of the failure arrival rate and the inter-arrival times of both duplicate and unique failures. Distributions of the time to repair all failures and duplicates are also provided.

It was found that 68% of the panics reported were software problems. Forty-three percent of the software failures were duplicates. The device drivers contributed the most

to the number of failures. These failures were also very common because many cases were reported. Failures were most commonly detected by triggers in the machine-level virtual memory code area. Sixty-two percent of the errors were detected very near to their source, while the other 38% percent showed some propagation. We also saw that the insertion of thousands of assertions in the OS improved error detection by detecting approximately 40% of all of the errors that displayed a panic string. Most of the failures were caused by missing exception-checking statements. On many occasions, wrong pointer addresses caused the failure. Using the above data, it can be concluded that there has been room for improvement of the OS robustness, as 38% of the faults incurred propagation before detection. Modules such as device drivers and the memory subsystem should be tested rigorously because most faults were present in these modules, and the errors were visible.

Inter-arrival times between the unique failures and the time between the duplicate faults were distributed with a mean of 20 and 85 days, respectively. The repair times of all failures were distributed with a mean of 40 days. The repair of the duplicates was faster (mean of 29 days) due mainly to fast identification of the faults using similar symptoms and to the availability of a fix. The availability of a core image also facilitates the repair process. It was observed that failures that recorded the core image had a higher probability of getting fixed. A study of the age (up-time) of the system when it failed showed that most of the system's ages were very low, suggesting that failures occur more often after a system upgrade or occur in succession due to the same fault.

## 6.2 Future Work

This research has opened a door in the area of research for understanding failures in the UNIX environment. Despite the universal use of UNIX-based systems in both industry and academia, not much research has been done on understanding the failure mechanism and behavior of a UNIX-based environment.

Continuing with the pursuit of developing a better understanding of failures in a UNIX-based system, a monitoring tool was implemented to collect failure logs from the CRHC Sun network in January 1995. This tool, which is written in *Perl*, is responsible for the collection of system logs from 64 SUN workstations in the CRHC (Center for Reliable and High Performance Computing) network. SunOS, which is based on the BSD UNIX, logs all of the console messages in a file called *messages* in the `var-adm` directory. Each week this message file is collected by the monitoring tool. At the end of the measurement period, the tool collates all of the messages based on the machine name and filters out the non-error messages.

Using the above tool, we monitored the CRHC network for eight months (January 95 to August 95). These data will be analyzed and classified in the near future. Preliminary results show that classification of failures can be done based on the client/server and the network. Because time information is also available, some timing distributions can also be generated.

# REFERENCES


[1] S. E. Butner and R. K. Iyer, "A statistical study of reliability and system load at SLAC," *Proc. 10th Int. Symp. Fault-Tolerant Computing*, Oct. 1980, pp. 207-209.

[2] X. Castillo and D. P. Siewiorek, "A comparable hardware/software reliability model," Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, 1981.

[3] R. Chillarege and R. K. Iyer, "Fault latency in memory– an experimental study on VAX 11/780," *Proc. 16th Int. Symp. Fault-Tolerant Computing*, 1986, pp. 258-263.

[4] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification–a concept for in-process measurement," *IEEE Trans. Softw. Eng.*, vol. 18 no. 11, Nov. 1992, pp. 943-955.

[5] R. Chillarege, S. Biyani, and J. Rosenthal, "Measurement of failure rate in widely distributed software," *Int. Symp. Fault-Tolerant Computing,* 1995, pp. 424-433.

[6] J. B. Dugan, "Correlated hardware failures in redundant systems," *Proc. IFIP Working Conf. Dependable Computing for Critical Applications*, 1991.

[7] J. Gray, "A census of Tandem system availability between 1985 and 1990," *IEEE Trans. Reliab.*, vol. 39, no. 4 Oct. 1990, pp. 409-418.

[8] J. P. Hansen and D. P. Siewiorek, "Models of time coalescence in event-logs," *Proc. 22nd Int. Symp. Fault-Tolerant Computing*, 1989, pp. 340-347.

[9] M. C. Hsueh and R. K. Iyer, "A measurement-based model of software reliability in a production environment," *Proc. 11th Int. Computer Software & Application Conference,* 1987, pp. 354-360.

[10] M. C. Hsueh, R. K. Iyer, and K. S. Trivedi, "Performability modeling based on real data: a case study," *IEEE Trans. Comput.*, vol. 37, no. 4, Apr. 1988, pp. 478-484.

[11] M. C. Hsueh, "Distributed multicomputer system availability based on measurement: a case study," *Proc. 10th Int. Phoenix Conference on Computers and Communications*, 1991, pp. 78-84.

[12] R .K. Iyer and D. J. Rossetti,"A statistical load dependency model for CPU errors at SLAC," *Proc. 12th Int. Symp. Fault-Tolerant Computing*, 1982, pp. 363-372.

[13] R. K. Iyer and P. Velardi, "Hardware-related software errors: measurement and analysis," *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 2, Feb. 1985, pp. 223-231.

[14] R. K. Iyer and D. J. Rossetti, "Effect of system workload on operating system reliability: a study on IBM 3081," *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 12, Dec. 1985, pp. 1438-1448.

[15] R. K. Iyer, D. J. Rosetti, and M. C. Hsueh, "Measurement and modeling of computer reliability as affected by system activity," *ACM Trans. Comput. Syst.*, vol. 4, no. 3, Aug. 1986, pp. 214-237.

[16] R. K. Iyer, L. T. Young, and P. V. K. Iyer, "Automatic recognition of intermittent failures: an experimental study of field data," *IEEE Trans. Comput.*, vol. 39, no. 4, Apr. 1990, pp. 525-537.

[17] R. K. Iyer and D. Tang, "Experimental analysis of computer system dependability," *University of Illinois, Urbana-Champaign, Tech. Rep. UILU-ENG-93-2227*, Sept. 1993.

[18] D. Jewett, "Integrity S2: A Fault Tolerant UNIX Platform," *Proc. Int. Symp. Fault-Tolerant Computing*, 1991.

[19] I. Lee, R. K. Iyer, and D. Tang, "Error/failure analysis using event logs from fault tolerant systems," *Proc. 21st Int. Symp. Fault-Tolerant Computing*, June 1991, pp. 10-17.

[20] I. Lee and R. K. Iyer, "Analysis of software halts in the Tandem GUARDIAN operating system," *Proc. Int. Symp. Software Reliability Engineering*, 1992.

[21] I. Lee, D. Tang, R. K. Iyer, and M. C. Hsueh, "Measurement-based evaluation of operating system fault tolerance," *IEEE Trans. Reliab.*, vol. 42, no. 2, Jun. 1993, pp. 238 -249.

[22] I. Lee and R. K. Iyer, "Faults, symptoms, and software fault tolerance in Tandem GUARDIAN90 operating system," *Proc. 23rd Int. Symp. Fault-Tolerant Computing*, 1993, pp. 20-29.

[23] I. Lee, R. K. Iyer, and Abhay Mehta, "Identifying software problems using symptoms," *Proc. 24th Int. Symp. Fault-Tolerant Computing*, 1994, pp. 320-329.

[24] I. Lee and R. K. Iyer, "Software Dependability in the Tandem GUARDIAN System," *IEEE Trans. on Softw. Eng.*, May 1995.

[25] T. T. Lin, and D. P. Siewiorek, "Error log analysis: statistical modeling and heuristic trend analysis," *IEEE Trans. Reliab.*, vol. 39, no. 4, Oct. 1990, pp. 419-432.

[26] B. Littlewood, S. Brocklehurst, T. Olovsson, and E. Jonsson, "On measurement of operational security," *IEEE Aerosp. Electron. Syst. Mag.*, vol. 9, no. 10, Oct. 1994, pp. 7-16.

[27] B. Littlewood, "Theories of software reliability: how good are they and how can they be improved?" *IEEE Trans. Softw. Eng.*, vol. SE-6, no. 5, Sept. 1980.

[28] M. R. Lyu, J. H. Chen, and A. Avizienis, "Software diversity metrics and measurements," *Proc. 16th Int. Computer Software and Applications Conference*, Sept. 1992, pp. 69-78.

[29] S. R. McConnel, D. P. Siewiorek, and M. M. Tsao, "The measurement and analysis of transient errors in Digital computer system," *Proc. 9th Int. Symp. Fault-Tolerant Computing*, 1979, pp. 67-70.

[30] J. G. McGough and F. L. Swern, "Measurement of fault latency in a digital avionic mini processor," NASA Contractor Rep.Tech. Rep. 3651, Jan 1983.

[31] R. A. Maxion, "Anomaly detection for diagnosis," *Proc. 20th Int. Symp. Fault-Tolerant Computing*, Jun. 1990, pp. 20-27,

[32] R. A. Maxion and F. E. Feather, "A case study of ethernet anomalies in a distributed computing environment," *IEEE Trans. Reliab.*, vol. 39, no. 4, Oct. 1990, pp. 433-443.

[33] R. A. Maxion and R. T. Olszewski, "Detection and discrimination of injected network faults," *Proc. 23rd Int. Symp. Fault-Tolerant Computing*, 1993, pp. 198-207.

[34] S. Mourad and D. Andrews, "On the reliability of the IBM MVS/XA operating system," *IEEE Trans. Softw. Eng.*, vol. SE-13, no. 10, Oct. 1987, pp. 1135-1139.

[35] J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability,* New York, NY: McGraw-Hill, 1990.

[36] D. P. Siewiorek, V. Kini, H. Mashburn, S. R. McConnel, and M. Tsao, "A case study of C.mmp, Cm, and C.vmp: part I: experience with fault tolerance in multiprocessor systems," *Proc. IEEE*, vol. 66, no. 10, Oct. 1978, pp. 1178-1199.

[37] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability – a study of field failures in operating systems," *Proc. 21st Int. Symp. Fault-Tolerant Computing*, June 1991.

[38] M. Sullivan and R. Chillarege, "A comparison of software defects in database management systems and operating systems," *Int. Symp. Fault-Tolerant Computing,* 1992, pp. 475-484.

[39] D. Tang, R. K. Iyer, and Sujatha Subramani, "Failure analysis and modeling of a VAXcluster system," *Proc. Int. Symp. Fault-Tolerant Computing*, 1990, pp. 244-251.

[40] D. Tang and R. K. Iyer, "Impact of correlated failures on dependability in a VAXcluster system," *Proc. 2nd IFIP Working Conf. Dependable Computing for Critical Applications*, 1991.

[41] D. Tang and R. K. Iyer, "Analysis of the VAX/VMS error logs in multicomputer environment environments - a case study of software dependability," *Int. Symp. Software Reliability Engineering,* 1992, pp. 216-226.

[42] D. Tang and R. K. Iyer, "Analysis and modeling of correlated failures in multicomputer systems," *IEEE Trans. Comput.*, vol. 41, no. 5, May 1992, pp. 567-577.

[43] D. Tang and R. K. Iyer, "Dependability measurement and modeling of a multicomputer systems," *IEEE Trans. Comput.*, vol. 42, no. 1, May 1992, pp. 62-75.

[44] M. M. Tsao and D. P. Siewiorek, "Trend analysis on system error files," *Proc. 13th Int. Symp. Fault-Tolerant Computing*, 1983. pp. 116-119.

[45] D. J. Taylor, D. E. Morgan, and J. Black, "Redundancy in data structures: improving software fault tolerance," *IEEE Trans. Softw. Eng.*, vol. SE-6, no. 6, Nov. 1980, pp. 585-594.

[46] T. A. Thayer, M. Lipow, and E. C. Nelson, *Software Reliability,* New York: Elsevier North-Holland, 1978.

[47] P. Velardi and R. K. Iyer, "A study of software failures and recovery in the MVS operating system," *IEEE Trans. Comput.*, vol. C-33, no. 6, Jun. 1984, pp. 564-568.

[48] A. S. Wein and A. Sathaye, "Validating complex computer system availability models," *IEEE Trans. Reliab.*, vol. 39, no. 4, Oct. 1990, pp. 468-479.

[49] M. H. Woodbury and K. G. Shin, "Measurement and analysis of workload effects on fault latency in real-time systems," *IEEE Trans. Softw. Eng.*, vol. 16, no. 2, Feb. 1990, pp. 212-216.